



Tema 2. Introducción a WPF

Desarrollo de Interfaces
DAM – IES Doctor Balmis



Javier Catalá

INDICE

1. Introducción
2. Controles
3. Eventos
4. Entrada (Ratón y Teclado)
5. Layout
6. Creación dinámica de controles
7. Estilos
8. Árbol lógico y árbol visual

1. Introducción

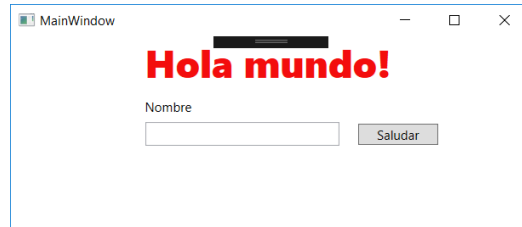
- WPF - Windows Presentation Foundation
- Tecnología de Microsoft que permite el desarrollo de aplicaciones visualmente atractivas
- Animaciones, gráficos 3D, documentos,...
- Introducida con el .NET Framework 3.0 (2006)
- Separa la IU de la lógica de negocio
- Introduce el lenguaje XAML

La tecnología que vamos a utilizar durante el curso es WPF. Microsoft introdujo esta tecnología junto con el sistema operativo Vista, con la intención de ofrecer a los desarrolladores un framework para crear aplicaciones visualmente más atractivas que las que se pueden desarrollar con WinForms.

Una de sus principales características es la utilización del lenguaje XAML, derivado de XML, para definir la interfaz de usuario de la aplicación. En la actualidad, este lenguaje se utiliza en otras tecnologías como UWP o Xamarin.

1. Introducción

```
<Window x:Class="WpfApp1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:WpfApp1"
        Title="MainWindow" Height="207.2" Width="489.6">
    <Grid>
        <TextBlock
            Text="Nombre"
            Margin="126,58,0,0" />
        <TextBox
            x:Name="nombreTextBox"
            Margin="126,81,179.2,77.6" />
        <Button
            x:Name="saludarButton"
            Content="Saludar"
            Margin="324,81,84.2,76.6"
            Width="75" Height="20"
            Click="SaludarButton_Click"/>
        <TextBlock
            x:Name="saludoTextBlock"
            Text="Hola mundo!"
            Margin="126,0,0,0"
            Foreground="#FFF30D0D"
            FontSize="36"
            FontFamily="Segoe UI Black"/>
    </Grid>
</Window>
```



4

Tema 2. Introducción a WPF

En la diapositiva se muestra el código XAML de una aplicación “Hola Mundo” en WPF.

Como se puede ver, hay una etiqueta *Window* que engloba todo el contenido de la ventana. Dentro de esta etiqueta se anidan las etiquetas del resto de componentes o controles que forman la ventana (en este caso *Grid*, *TextBlock*, *TextBox* o *Button*). Cada uno de estos controles se configura por medio de sus propiedades. Algunas propiedades son comunes a la mayoría de los controles, y otras son particulares para un determinado tipo de control.

La propiedad *x:Name* asigna al control un nombre para poder hacer referencia a él.

1. Introducción

```
using System.Windows;

namespace WpfApp1
{
    /// <summary>
    /// Lógica de interacción para MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void SaludarButton_Click(object sender, RoutedEventArgs e)
        {
            saludoTextBlock.Text = "Hola " + nombreTextBox.Text + "!";
        }
    }
}
```

Solución "WpfApp1" (1 proyecto)

- WpfApp1
 - Properties
 - Referencias
 - App.config
 - App.xaml
 - MainWindow.xaml
 - MainWindow.xaml.cs

Código trasero
(code behind)

5

Tema 2. Introducción a WPF

Las ventanas en WPF se definen en un fichero con extensión *.xaml*, como puede apreciarse en la diapositiva. Además, la ventana tiene un fichero de código asociado, conocido como código trasero (o *code behind* en inglés).

En el código trasero escribiremos el código C# necesario para el funcionamiento de la aplicación, principalmente el código de los manejadores de eventos. En el ejemplo se incluye el manejador del evento *Clic* del botón.

La ventana es en realidad una clase que deriva de la clase *Window*, y cuenta con un constructor para la inicialización de la ventana. En este constructor podemos añadir código que se deba ejecutar al crear la ventana.

2. Controles

Control = **Comportamiento** + Apariencia

Personalizar apariencia:

① Propiedades



② Contenido anidado



③ Plantilla del control



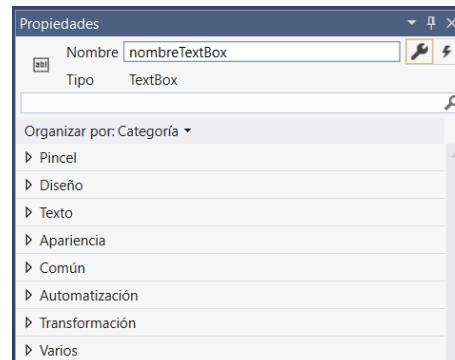
Un control en WPF es la combinación de un determinado comportamiento con una apariencia visual. Una de las características de WPF es que la apariencia de un control es totalmente personalizable por el desarrollador. Existen tres vías para ello:

1. A través de las propiedades del control
2. Incluyendo contenido dentro del control (si lo permite)
3. Modificando la plantilla del control

El comportamiento de un control no se puede modificar. Lo que sí que permite WPF es crear controles nuevos con el comportamiento que necesitemos, pudiendo para ello partir de algún control existente.

2. Controles

```
<Grid>
  <TextBlock
    Text="Nombre"
    Margin="126,58,0,0" />
  <TextBox
    x:Name="nombreTextBox"
    Margin="126,81,179.2,77.6" />
  <Button
    x:Name="saludarButton"
    Content="Saludar"
    Margin="324,81,84.2,76.6"
    Width="75" Height="20"
    Click="SaludarButton_Click"/>
  <TextBlock
    x:Name="saludoTextBlock"
    Text="Hola mundo!"
    Margin="126,0,0,0"
    Foreground="#FFF30D0D"
    FontSize="36"
    FontFamily="Segoe UI Black"/>
</Grid>
```



Cuando utilizamos Visual Studio para desarrollar aplicaciones WPF tenemos a nuestra disposición un panel de propiedades para configurar los diferentes controles de la ventana. Las propiedades se encuentran agrupadas por categorías para facilitar su localización, y en función del tipo de control seleccionado encontraremos unas propiedades u otras.

Aunque en ocasiones el panel de propiedades puede ser de utilidad, lo más habitual es trabajar directamente en el código XAML.

2. Controles

```
<Button Foreground="Black">  
    <Button.Background>  
        <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">  
            <GradientStop Color="Black" Offset="0"/>  
            <GradientStop Color="White" Offset="1"/>  
        </LinearGradientBrush>  
    </Button.Background>  
    Aceptar  
</Button>
```

Propiedad como atributo XML

Propiedad como subelemento XML

Propiedad por defecto (Content para el botón)

Cuando definimos las propiedades de un control directamente en XAML tenemos distintas opciones. La más simple y más común es establecerla directamente como un atributo del elemento XML que define el control.

Sin embargo, en algunas ocasiones el valor de una propiedad no se podrá expresar como un atributo XML, y será necesario utilizar la notación de elemento XML.

También es importante destacar que cada control tendrá una propiedad por defecto, que podremos establecer con el contenido situado entre las etiquetas de apertura y cierre del control, sin necesidad de hacer referencia al nombre de la propiedad.

2. Controles



```
<Button>  
    <Image Source="prueba.png" />  
</Button>
```














































Muchos controles pueden tener otro elemento como contenido. Se les conoce como *Content Controls* y su propiedad por defecto es *Content*.

WPF permite que el contenido de algunos controles (como el control *Button*) sea a su vez otro elemento XAML. A estos controles se les conoce como *Content Controls*. En el ejemplo se muestra un botón cuyo contenido es un control de imagen.

Hay que tener en cuenta que los *Content Controls* pueden albergar en su interior como máximo un control.

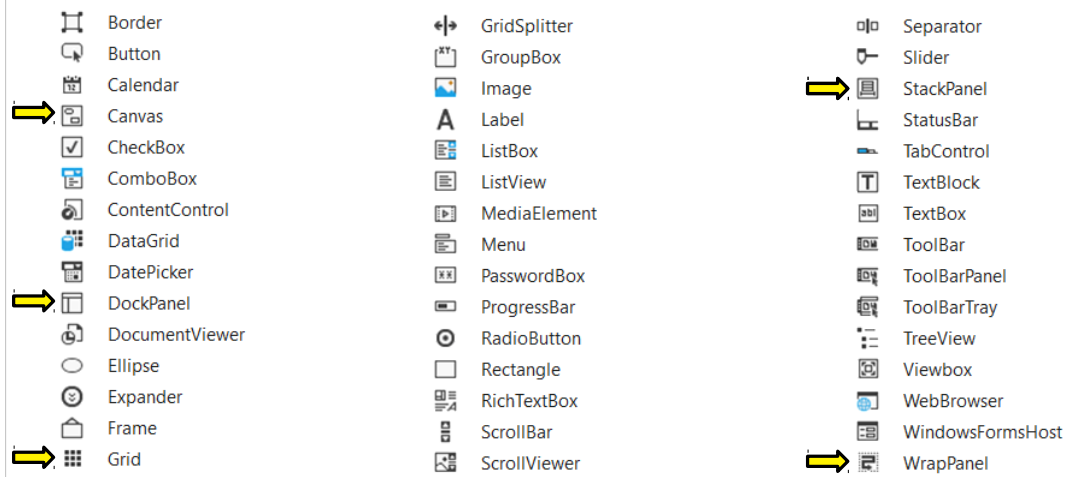
Las ventanas son otro ejemplo de *Content Control* y por tanto solo pueden albergar un control en su interior. Obviamente, una ventana necesitará en la mayor parte de los casos más de un control, y para salvar esta limitación utilizaremos controles contenedores, que veremos más adelante.

2. Controles

 Border	 GridSplitter	 Separator
 Button	 GroupBox	 Slider
 Calendar	 Image	 StackPanel
 Canvas	 Label	 StatusBar
 CheckBox	 ListBox	 TabControl
 ComboBox	 ListView	 TextBlock
 ContentControl	 MediaElement	 TextBox
 DataGrid	 Menu	 ToolBar
 DatePicker	 PasswordBox	 ToolBarPanel
 DockPanel	 ProgressBar	 ToolBarTray
 DocumentViewer	 RadioButton	 TreeView
 Ellipse	 Rectangle	 Viewbox
 Expander	 RichTextBox	 WebBrowser
 Frame	 ScrollBar	 WindowsFormsHost
 Grid	 ScrollViewer	 WrapPanel

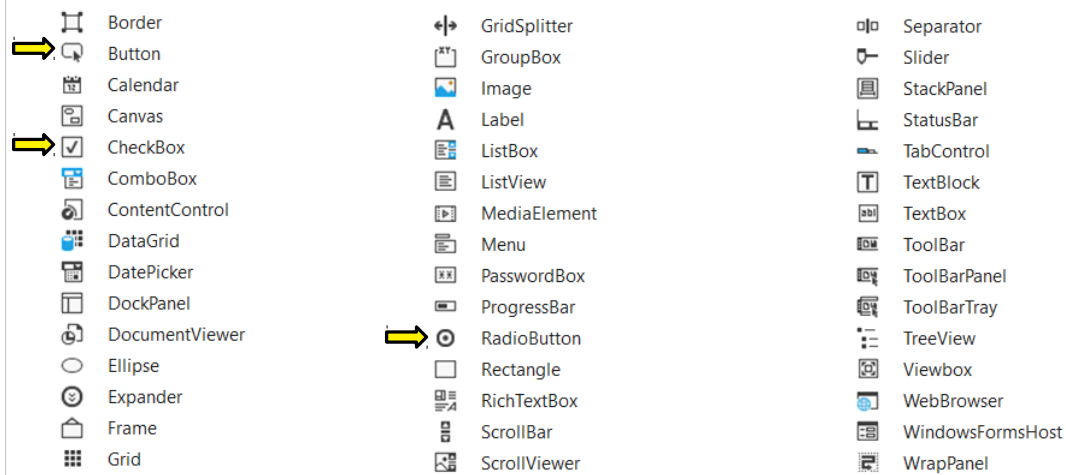
En la diapositiva se muestra un listado de los controles WPF disponibles en el cuadro de herramientas de Visual Studio. Aunque están la mayoría de los controles existentes, hay que tener en cuenta que hay algunos controles que no están en este listado pero pueden ser utilizados directamente en XAML (por ejemplo el *RepeatButton*).

2. Controles - Layout



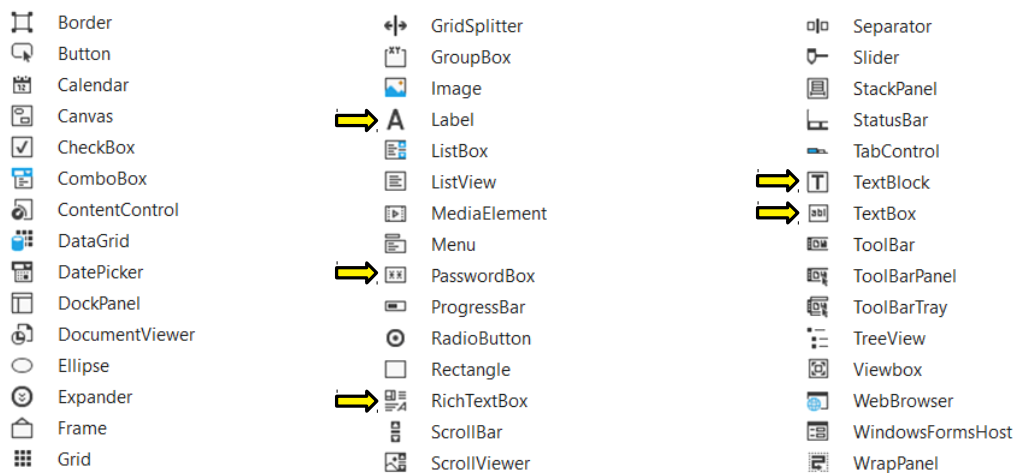
En esta dispositiva se han señalado los controles relacionados con el *layout*. Son los controles que nos ayudaran a organizar el resto de controles en la ventana. También se les conoce como contenedores, y los veremos con detalle dentro de este mismo tema.

2. Controles - Botones



Dentro de la categoría de controles, encontramos los botones de acción, los botones de radio y las casillas de verificación.

2. Controles - Texto

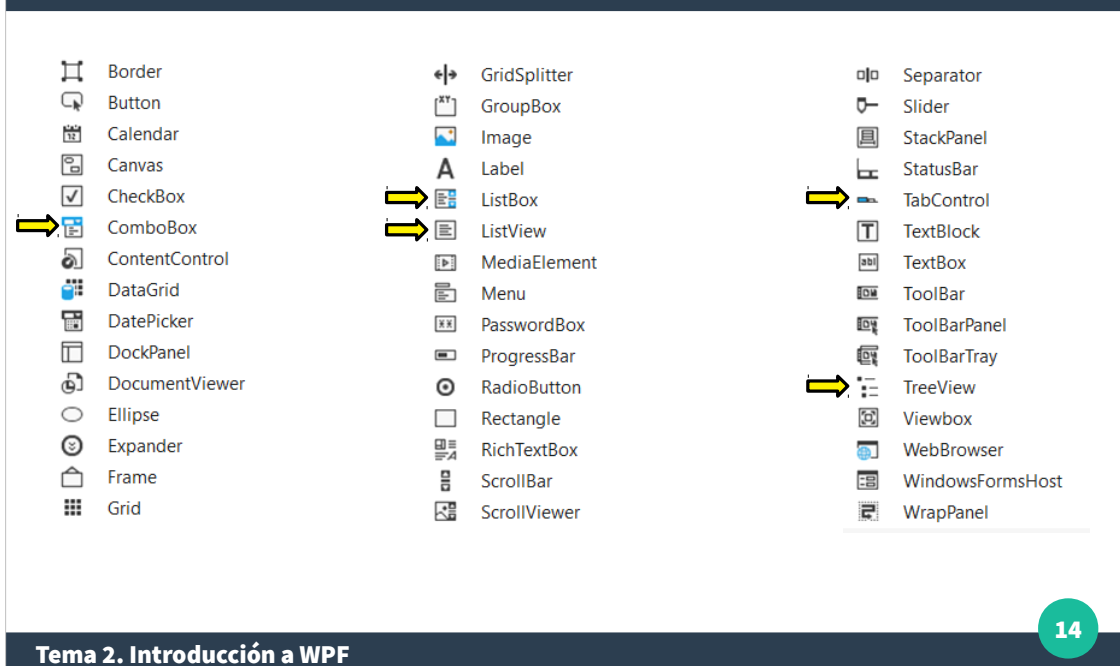


Los controles relacionados con texto los podemos dividir en dos grupos. Por un lado, los controles que permiten mostrar texto en la ventana, y por otro los que permiten al usuario introducir texto.

En la primera categoría tenemos los controles *TextBlock* y *Label*. En un principio los dos controles tienen una funcionalidad similar, pero iremos viendo a lo largo del curso las características que los diferencian (por ejemplo, *Label* es *Content Control* y *TextBlock* no).

En la segunda categoría tenemos los cuadros de texto (*TextBox*), los cuadros de texto para contraseñas (*PasswordBox*) y los que permiten texto con formato enriquecido (*RichTextBox*).

2. Controles - Listas
















































Este grupo de controles permiten mostrar información en forma de lista con varios elementos.

El *ComboBox* y el *ListBox* son los más básicos y son muy similares. La principal diferencia es que el *ComboBox* es desplegable y permite al usuario introducir un valor no presente en el listado.

El *ListView* permite mostrar un listado de elementos con diferentes vistas (similar al explorador de archivos de Windows). El *TreeView* se utiliza para mostrar elementos con relación jerárquica, como por ejemplo un árbol de directorios.

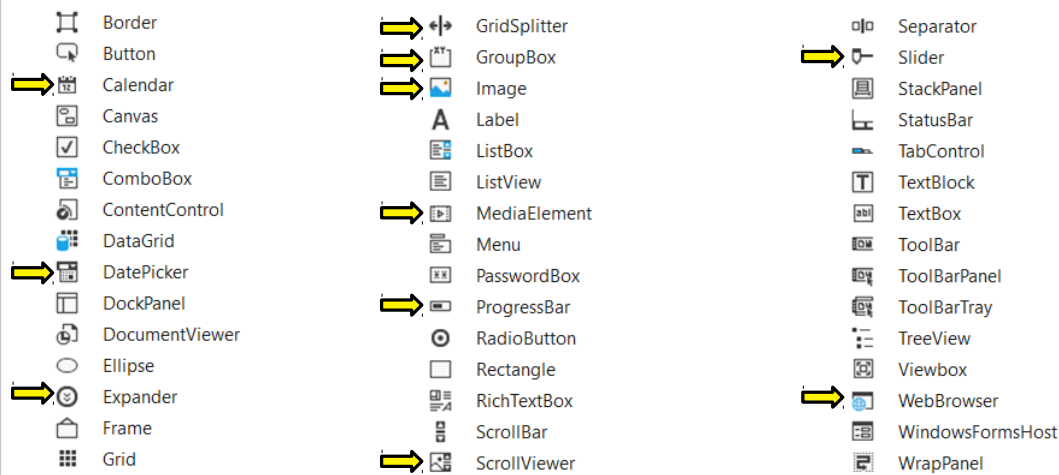
También se incluye en esta categoría el *TabControl*, que nos permite organizar los controles de una ventana utilizando pestañas.

2. Controles - Aplicaciones

 Border	 GridSplitter	 Separator
 Button	 GroupBox	 Slider
 Calendar	 Image	 StackPanel
 Canvas	 Label	 StatusBar
 CheckBox	 ListBox	 TabControl
 ComboBox	 ListView	 TextBlock
 ContentControl	 MediaElement	 TextBox
 DataGrid	 Menu	 ToolBar
 DatePicker	 PasswordBox	 ToolBarPanel
 DockPanel	 ProgressBar	 ToolBarTray
 DocumentViewer	 RadioButton	 TreeView
 Ellipse	 Rectangle	 Viewbox
 Expander	 RichTextBox	 WebBrowser
 Frame	 ScrollBar	 WindowsFormsHost
 Grid	 ScrollViewer	 WrapPanel

Esta categoría contiene controles que solemos encontrar en las aplicaciones habitualmente, como menús, barras de herramientas o barra de estado (la barra que suele aparecer en la parte inferior de la ventana con información sobre la aplicación).

2. Controles - Otros



Además de los controles comentados hasta ahora WPF incorpora muchos otros controles, como por ejemplo:

- *Calendar: permite insertar un calendario en la aplicación*
- *DatePicker: permite al usuario seleccionar una fecha*
- *Expander: se utiliza para mostrar u ocultar controles*
- *GridSplitter: permite al usuario redimensionar*
- *GroupBox: agrupa controles con un marco y una etiqueta*
- *Image: se utiliza para insertar una imagen*
- *MediaElement: reproduce vídeo y audio*
- *ProgressBar: barra de progreso para indicar avance*
- *ScrollViewer: añade barras de desplazamiento*
- *Slider: seleccionar un valor numérico en un rango*
- *WebBrowser: permite incrustar contenido web*

2. Controles

Button	
Propiedades	Eventos
Content	Click
ClickMode	
IsDefault	
IsCancel	
IsEnabled	

Propiedades:

- *Content*: el contenido que se mostrará dentro del botón
- *ClickMode*: indica cuando debe producirse el evento Click
- *IsDefault*: indica si el botón se asocia a la tecla Intro
- *IsCancel*: indica si el botón se asocia a la tecla Esc
- *IsEnabled*: indica si el botón está habilitado o no

Eventos:

- *Click*: se genera cuando el usuario presiona el botón

2. Controles

TextBox	
Propiedades	Eventos
Text	TextChanged
AcceptsReturn	
TextWrapping	
IsReadOnly	

Propiedades:

- *Text*: contiene el texto del TextBox
- *AcceptReturn*: indica si es posible utilizar la tecla Intro dentro del TextBox
- *TextWrapping*: establece el ajuste de línea
- *IsReadOnly*: indica si el TextBox es de solo lectura

Eventos:

- *TextChanged*: se produce cada vez que se modifica el texto del TextBox

2. Controles

Image	
Propiedades	Eventos
Source	
Stretch	

Propiedades:

- *Source: establece el origen de la imagen a mostrar*
- *Stretch: indica como la imagen debe ajustarse en su contenedor*

2. Controles

CheckBox	
Propiedades	Eventos
Content	Checked
IsChecked	Unchecked
IsThreeState	Indeterminate

Propiedades:

- *Content*: establece el contenido que se mostrará
- *IsChecked*: indica si el checkbox está marcado o no
- *IsThreeState*: indica si el checkbox admite el tercer estado

Eventos:

- *Checked*: se produce cuando se marca el checkbox
- *Unchecked*: se produce cuando se desmarca el checkbox
- *Indeterminate*: se produce cuando el el chekbox está en el tercer estado (estado indeterminado)

2. Controles

RadioButton	
Propiedades	Eventos
Content	Checked
IsChecked	Unchecked
GroupName	

Propiedades:

- *Content*: establece el contenido que se mostrará
- *IsChecked*: indica si el radiobutton está marcado
- *GroupName*: permite agrupar varios controles de este tipo

Eventos:

- *Checked*: se produce cuando se marca el control
- *Unchecked*: se produce cuando se desmarca el control

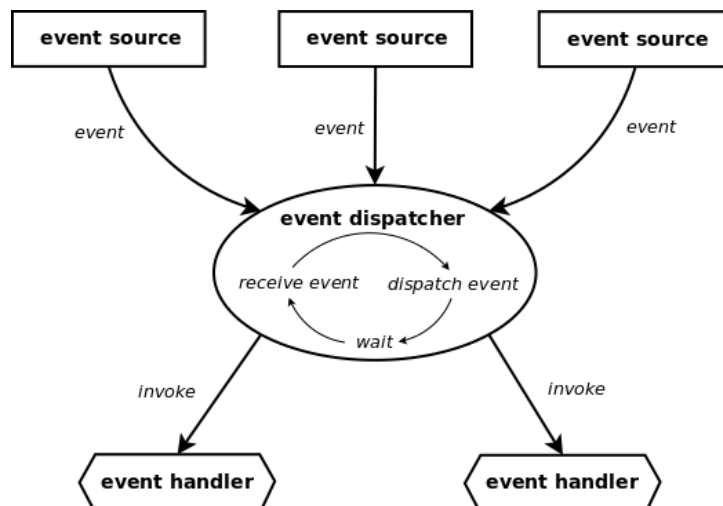
2. Controles

TextBlock	
Propiedades	Eventos
Text	
TextTrimming	
TextWrapping	
TextAlignment	

Propiedades:

- *Text*: contiene el texto que se mostrará en el TextBlock
- *TextTrimming*: indica el comportamiento del TextBlock cuando el texto no se puede mostrar en su totalidad
- *TextWrapping*: indica si se debe realizar ajuste de línea
- *TextAlignment*: indica el tipo de alineado

3. Eventos



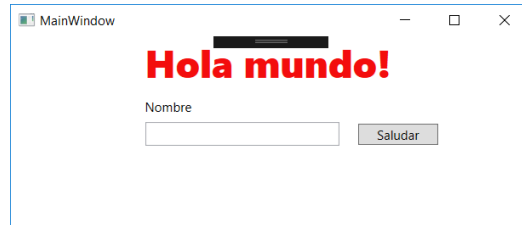
Como ya comentamos en el primer tema, la programación de aplicaciones con interfaz gráfica de usuario se basa en el paradigma orientado a eventos.

De esta forma, los distintos controles WPF pueden generar diferentes eventos, que son recibidos por el gestor de eventos del framework. Para cada evento, el gestor invoca a los manejadores (delegados) suscritos a cada evento.

Una parte importante del desarrollo de aplicaciones WPF consiste en conocer los diferentes eventos que generan los controles disponibles.

3. Eventos

```
<Window x:Class="WpfApp1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:WpfApp1"
        Title="MainWindow" Height="207.2" Width="489.6">
    <Grid>
        <TextBlock
            Text="Nombre"
            Margin="126,58,0,0" />
        <TextBox
            x:Name="nombreTextBox"
            Margin="126,81,179.2,77.6" />
        <Button
            x:Name="saludarButton"
            Content="Saludar"
            Margin="324,81,84.2,76.6"
            Width="75" Height="30"
            Click="SaludarButton_Click"/>
        <TextBlock
            x:Name="saludoTextBlock"
            Text="Hola mundo!"
            Margin="126,0,0,0"
            Foreground="#FFF30D0D"
            FontSize="36"
            FontFamily="Segoe UI Black"/>
    </Grid>
</Window>
```



Como ya vimos en el primer ejemplo, la forma de asociar un manejador a un evento es mediante una propiedad del elemento XAML asociado al control.

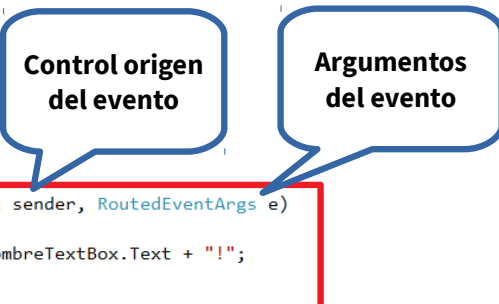
Visual Studio nos ofrece la posibilidad de crear el método por nosotros, con los parámetros adecuados al tipo de evento.

3. Eventos

```
using System.Windows;

namespace WpfApp1
{
    /// <summary>
    /// Lógica de interacción para MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void SaludarButton_Click(object sender, RoutedEventArgs e)
        {
            saludoTextBlock.Text = "Hola " + nombreTextBox.Text + "!";
        }
    }
}
```



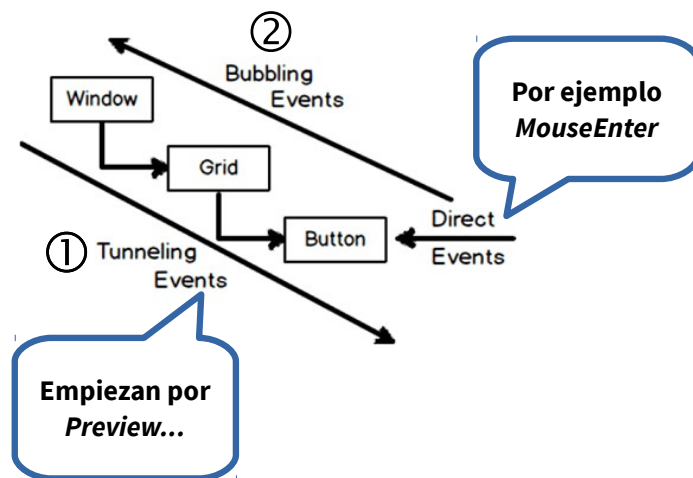
The diagram illustrates the parameters of the event handler method `SaludarButton_Click`. A red box highlights the method signature `private void SaludarButton_Click(object sender, RoutedEventArgs e)`. Two callout boxes are present: one labeled "Control origen del evento" pointing to the `sender` parameter, and another labeled "Argumentos del evento" pointing to the `e` parameter.

Los métodos manejadores de los eventos siempre tienen dos parámetros de entrada:

- El objeto que ha generado el evento (*sender*)
- Información relacionada con el evento (*e*)

La clase a utilizar para el segundo parámetro dependerá del tipo de evento asociado. En la documentación online de WPF se puede consultar dicha información.

3. Eventos



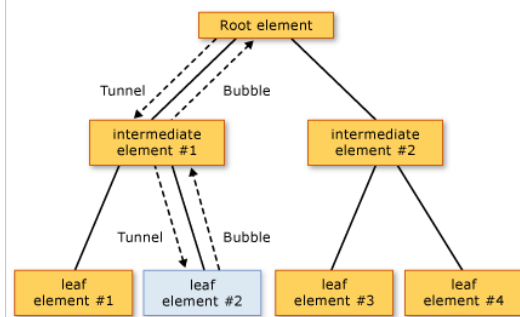
WPF sigue un modelo de eventos enrutados. Cuando se genera un evento en un control, el mismo evento se enruta por todos los controles que le anteceden en el árbol lógico.

Existen dos tipos diferentes de eventos, según el tipo de enrutado:

- Eventos de túnel: se genera primero el evento en la raíz, y se desciende hasta el control que lo origina. Su nombre suele empezar por *Preview*.
- Eventos de burbuja: se genera primero en el control origen, y de ahí se sube hasta el elemento raíz.

Algunos eventos (llamados eventos directos) no son enrutados, y solo se generan en el control origen.

3. Eventos



1. `PreviewMouseDown` (tunnel) on root element.
2. `PreviewMouseDown` (tunnel) on intermediate element #1.
3. `PreviewMouseDown` (tunnel) on source element #2.
4. `MouseDown` (bubble) on source element #2.
5. `MouseDown` (bubble) on intermediate element #1.
6. `MouseDown` (bubble) on root element.

En la diapositiva se muestra el orden en el que se generarían los distintos eventos en los diferentes elementos en caso de una pulsación de ratón sobre el elemento señalado en azul.

Como se puede apreciar, en primer lugar se producen los eventos de tipo túnel, desde el elemento raíz hacia abajo. A continuación, se desencadenan los eventos de burbuja desde el control que genera el evento hasta el elemento raíz.

3. Eventos

```
private void SaludarButton_Click(object sender, RoutedEventArgs e)
{
    saludoTextBlock.Text = "Hola " + nombreTextBox.Text + "!";

    e.Handled = true;
}
```

Con esta asignación paramos
el enrutado del evento

Si en alguna ocasión nos interesa parar el enrutado del evento, podemos hacerlo dando el valor `True` a la propiedad *Handled* del parámetro `e` en el manejador de eventos.

4. Entrada - Ratón

Event	Routing	Meaning
GotMouseCapture	Bubble	Element captured the mouse.
LostMouseCapture	Bubble	Element lost mouse capture.
MouseEnter	Direct	Mouse pointer moved into element.
MouseLeave	Direct	Mouse pointer moved out of element.
PreviewMouseLeftButtonDown, MouseLeftButtonDown	Tunnel, Bubble	Left mouse button pressed while pointer inside element.
PreviewMouseLeftButtonUp, MouseLeftButtonUp	Tunnel, Bubble	Left mouse button released while pointer inside element.
PreviewMouseRightButtonDown, MouseRightButtonDown	Tunnel, Bubble	Right mouse button pressed while pointer inside element.
PreviewMouseRightButtonUp, MouseRightButtonUp	Tunnel, Bubble	Right mouse button released while pointer inside element.
PreviewMouseDown, MouseDown	Tunnel, Bubble	Mouse button pressed while pointer inside element (raised for any mouse button).
PreviewMouseUp, MouseUp	Tunnel, Bubble	Mouse button released while pointer inside element (raised for any mouse button).
PreviewMouseMove, MouseMove	Tunnel, Bubble	Mouse pointer moved while pointer inside element.
PreviewMouseWheel, MouseWheel	Tunnel, Bubble	Mouse wheel moved while pointer inside element.
QueryCursor	Bubble	Mouse cursor shape to be determined while pointer inside element.

En la diapositiva se recogen los eventos relacionados con el ratón que generan la mayoría de los controles WPF, indicándose además el tipo de enrutado del evento.

Como se puede apreciar, se puede controlar cualquier tipo de acción relacionada con el ratón. Llama la atención la ausencia de un evento *Click*. La razón es que esta acción del usuario es en realidad una combinación de eventos (pulsar y soltar el botón izquierdo del ratón) y por ello no aparece en el listado.

4. Entrada - Ratón

- **Eventos de alto nivel:**
 - `MouseDoubleClick` / `PreviewMouseDoubleClick` (clase *Control*)
 - `Click` (clase *ButtonBase*)
- **Propiedad *IsMouseOver***
- **Clase estática *Mouse*:**
 - Método *GetPosition*
 - Propiedad *DirectlyOver*
 - Propiedades *LeftButton*, *MiddleButton*, *RightButton*

Además de los eventos que hemos visto, existen dos eventos de alto nivel (formados por la combinación de otros eventos) introducidos por clases más específicas:

- *MouseDoubleClick/PreviewMouseDoubleClick*: introducido por la clase *Control*, representa el doble clic con el ratón
- *Click*: introducido por la clase *ButtonBase*, representa un clic con el ratón.

Además de todos los eventos relacionados con el ratón, los controles cuentan con la propiedad *IsMouseOver* para saber si el ratón está sobre ellos.

Por último, tenemos a nuestra disposición la clase estática *Mouse*, con métodos y propiedades relacionados con el ratón.

4. Entrada - Teclado

Event	Routing	Meaning
PreviewGotKeyboardFocus, GotKeyboardFocus	Tunnel, Bubble	Element received the keyboard focus.
PreviewLostKeyboardFocus, LostKeyboardFocus	Tunnel, Bubble	Element lost the keyboard focus.
GotFocus	Bubble	Element received the logical focus.
LostFocus	Bubble	Element lost the logical focus.
PreviewKeyDown, KeyDown	Tunnel, Bubble	Key pressed.
PreviewKeyUp, KeyUp	Tunnel, Bubble	Key released.
PreviewTextInput, TextInput	Tunnel, Bubble	Element received text input.

Como en el caso del ratón, también tenemos a nuestra disposición una gran cantidad de eventos para controlar las acciones del usuario con el teclado.

Algunos de los eventos están relacionados con el foco. El control que tiene el foco es el que recibe la entrada del teclado. Como se puede apreciar, algunos eventos distinguen entre el foco del teclado y el foco lógico. Cuando el usuario está utilizando nuestra aplicación, el mismo control tendrá el foco lógico y el foco de teclado. Sin embargo, cuando el usuario cambie de aplicación el control perderá el foco de teclado, pero mantendrá el foco lógico.

4. Entrada - Teclado

- Propiedad *IsFocused*
- Clase estática *Keyboard*:
 - Propiedad *Modifiers* (teclas Alt, Shift y Ctrl)
 - Métodos *IsKeyDown* / *IsKeyUp*
 - Propiedad *FocusedElement*
 - Método *Focus*

Como en el caso del ratón, además de los eventos vistos tenemos a nuestra disposición otras herramientas para trabajar con el teclado.

Por un lado, la propiedad *IsFocused* que tienen la mayoría de los controles nos indicará si el control en cuestión tiene el foco.

La clase estática *Keyboard* tiene métodos y propiedades relacionadas con el teclado. Por ejemplo, el método *Focus* (que permite asignar el foco a un elemento) o la propiedad *FocusedElement*, que permite acceder al elemento que tiene el foco.

5. Layout

- Diseño adaptable a cambios en el tamaño y configuración de la ventana
- Se basa en el uso de paneles:
 - StackPanel
 - WrapPanel
 - DockPanel
 - Grid
 - Canvas

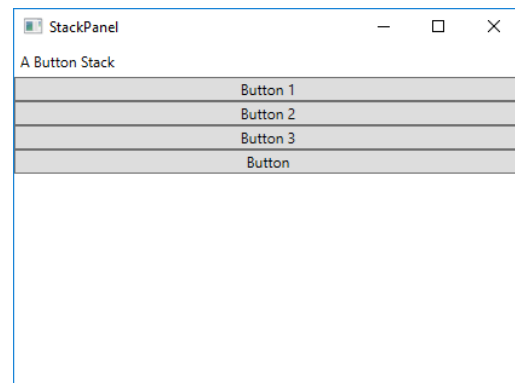
En este apartado vamos a ver las herramientas que WPF ofrece para organizar los elementos de una ventana. Es lo que se conoce como *layout*.

Todo el sistema de *layout* de WPF se basa en la idea de que el diseño de las ventanas se debe adaptar correctamente a los cambios en el tamaño de las mismas.

La organización de los controles se basa en la utilización de un grupo especial de elementos conocidos como paneles o contenedores, cuya función consiste en albergar otros controles en su interior.

5. Layout - StackPanel

```
<StackPanel>  
  <Label>A Button Stack</Label>  
  <Button>Button 1</Button>  
  <Button>Button 2</Button>  
  <Button>Button 3</Button>  
  <Button>Button</Button>  
</StackPanel>
```

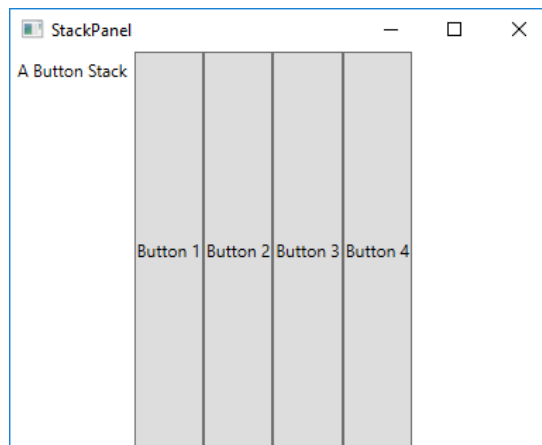


El contenedor más sencillo es el *StackPanel*. Los controles situados dentro este contenedor quedaran apilados de forma vertical, desde arriba hacia abajo.

Como se puede apreciar en el ejemplo, por defecto los controles ocuparán todo el espacio horizontal del *StackPanel*.

5. Layout - StackPanel

```
<StackPanel Orientation="Horizontal">  
  <Label>A Button Stack</Label>  
  <Button>Button 1</Button>  
  <Button>Button 2</Button>  
  <Button>Button 3</Button>  
  <Button>Button 4</Button>  
</StackPanel>
```



Aunque el *StackPanel* apila por defecto los controles en vertical, cuenta con la propiedad *Orientation* para cambiar este comportamiento.

Como podemos observar en la diapositiva, si configuramos el *StackPanel* para que apile de forma horizontal, los controles ocuparán todo el espacio vertical.

5. Layout - StackPanel

Layout Properties

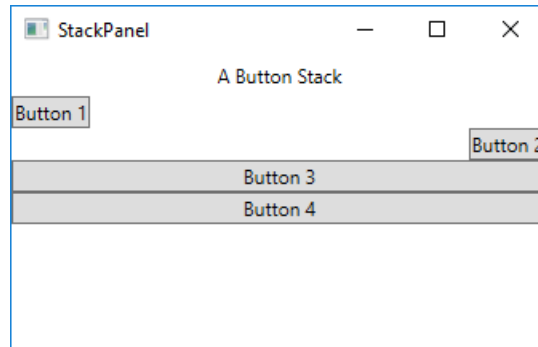
Name	Description
HorizontalAlignment	Determines how a child is positioned inside a layout container when there's extra horizontal space available. You can choose Center, Left, Right, or Stretch.
VerticalAlignment	Determines how a child is positioned inside a layout container when there's extra vertical space available. You can choose Center, Top, Bottom, or Stretch.
Margin	Adds a bit of breathing room around an element. The Margin property is an instance of the System.Windows.Thickness structure, with separate components for the top, bottom, left, and right edges.
MinWidth and MinHeight	Sets the minimum dimensions of an element. If an element is too large for its layout container, it will be cropped to fit.
MaxWidth and MaxHeight	Sets the maximum dimensions of an element. If the container has more room available, the element won't be enlarged beyond these bounds, even if the HorizontalAlignment and VerticalAlignment properties are set to Stretch.
Width and Height	Explicitly sets the size of an element. This setting overrides a Stretch value for the HorizontalAlignment or VerticalAlignment properties. However, this size won't be honored if it's outside of the bounds set by the MinWidth, MinHeight, MaxWidth, and MaxHeight.

Existen una serie de propiedades comunes a los controles muy importantes a la hora de establecer el *layout* de la aplicación:

- *HorizontalAlignment* y *VerticalAlignment* controlan la alineación horizontal y vertical del control en su contenedor. Su valor por defecto es *Stretch*.
- *Margin* establece el espacio entre el límite del control y el resto de controles a su alrededor.
- *Width* y *Height* permiten dar un tamaño concreto al control, independientemente del tamaño del contenedor.
- *MinWidth/MinHeight* y *MaxWidth/MaxHeight* permiten establecer unos límites inferiores y superiores al tamaño del control.

5. Layout - StackPanel

```
<StackPanel>  
  <Label HorizontalAlignment="Center">A Button Stack</Label>  
  <Button HorizontalAlignment="Left">Button 1</Button>  
  <Button HorizontalAlignment="Right">Button 2</Button>  
  <Button>Button 3</Button>  
  <Button>Button 4</Button>  
</StackPanel>
```

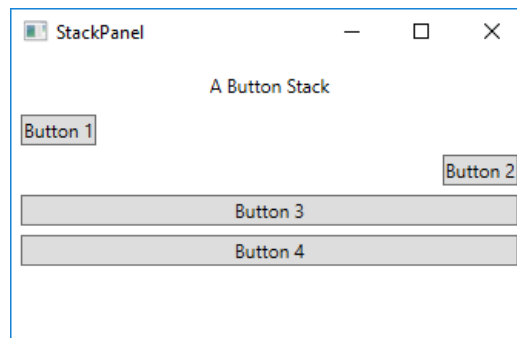


En esta diapositiva se muestra un ejemplo de uso de la propiedad *HorizontalAlignment*. Como se puede observar, al establecer un valor diferente a *Stretch* el ancho del control se ajustará al contenido del mismo.

También es importante destacar que el en caso de un *StackPanel* vertical, el uso de la propiedad *VerticalAlignment* no tendría ningún efecto por la propia naturaleza de este contenedor.

5. Layout - StackPanel

```
<StackPanel Margin="3">
  <Label Margin="3" HorizontalAlignment="Center">
    A Button Stack</Label>
  <Button Margin="3" HorizontalAlignment="Left">Button 1</Button>
  <Button Margin="3" HorizontalAlignment="Right">Button 2</Button>
  <Button Margin="3">Button 3</Button>
  <Button Margin="3">Button 4</Button>
</StackPanel>
```

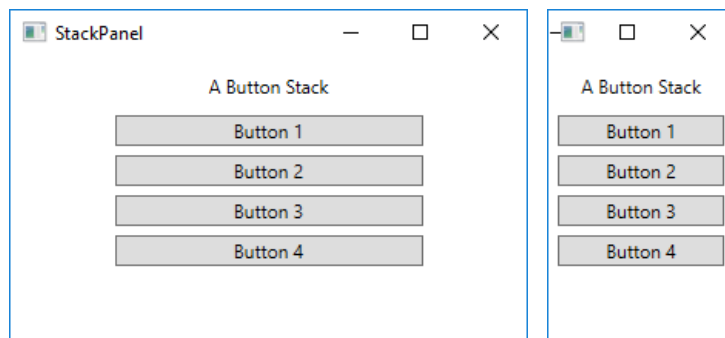


Este ejemplo es similar al de la diapositiva anterior, con la diferencia de que en este caso se ha utilizado la propiedad *Margin* para dejar un espacio entre los controles.

Comentar que es posible dar un valor diferente a cada uno de los cuatro márgenes (izquierda, arriba, derecha, abajo).

5. Layout - StackPanel

```
<StackPanel Margin="3">
  <Label Margin="3" HorizontalAlignment="Center">
    A Button Stack</Label>
  <Button Margin="3" MaxWidth="200" MinWidth="100">Button 1</Button>
  <Button Margin="3" MaxWidth="200" MinWidth="100">Button 2</Button>
  <Button Margin="3" MaxWidth="200" MinWidth="100">Button 3</Button>
  <Button Margin="3" MaxWidth="200" MinWidth="100">Button 4</Button>
</StackPanel>
```



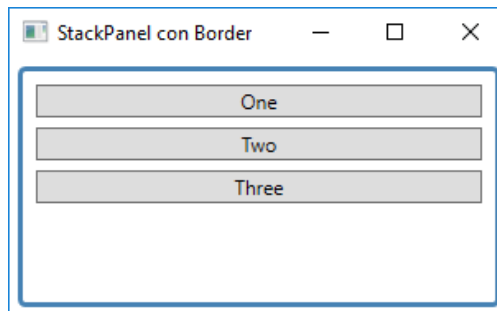
Este ejemplo muestra el uso de las propiedades para establecer un mínimo y un máximo en el ancho y alto del control.

Como se puede observar, al haber asignado un ancho máximo, el tamaño de los controles no aumentará a pesar del tamaño disponible en el contenedor.

De la misma forma, al haber establecido un ancho mínimo, el tamaño del control no descenderá de ese mínimo.

5. Layout – StackPanel con Border

```
<Border Margin="5" Padding="5" BorderBrush="SteelBlue" BorderThickness="3" CornerRadius="3">  
  <StackPanel>  
    <Button Margin="3">One</Button>  
    <Button Margin="3">Two</Button>  
    <Button Margin="3">Three</Button>  
  </StackPanel>  
</Border>
```



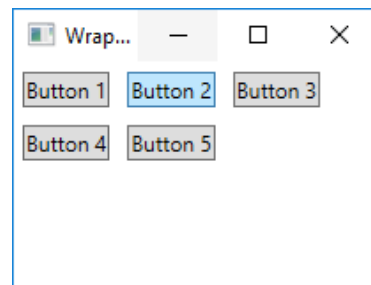
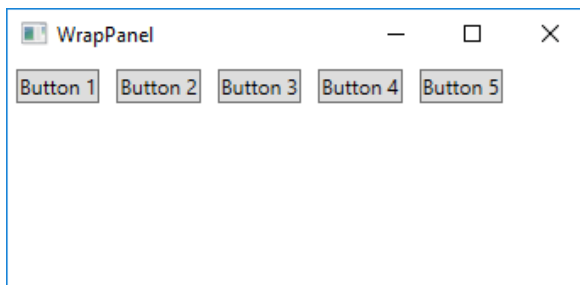
Una carencia que presentan todos los contenedores es la del borde. Ninguno de ellos cuenta con una propiedad para que se muestre un borde alrededor del control.

Para aportar esta funcionalidad WPF utiliza un elemento llamado *Border*, que permite dibujar un borde alrededor de cualquier control. Este elemento pertenece a una categoría de controles llamados decoradores.

El control *Border* dispone de diferentes propiedades para su configuración, como las que se muestran en el ejemplo.

5. Layout – WrapPanel

```
<WrapPanel>  
  <Button Margin="5">Button 1</Button>  
  <Button Margin="5">Button 2</Button>  
  <Button Margin="5">Button 3</Button>  
  <Button Margin="5">Button 4</Button>  
  <Button Margin="5">Button 5</Button>  
</WrapPanel>
```



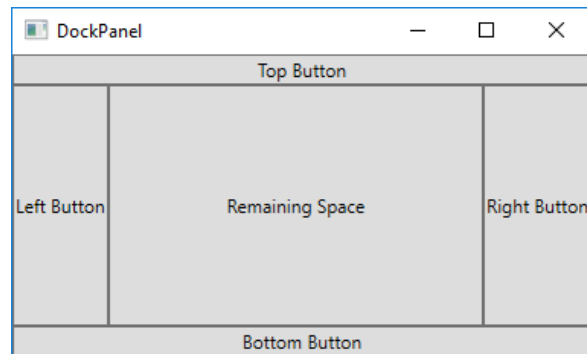
El *WrapPanel* es un contenedor en el que los controles se sitúan de forma secuencial. Si los controles no caben en una sola fila, automáticamente se reajustan en las filas necesarias.

Como el *StackPanel*, el *WrapPanel* dispone de la propiedad *Orientation*, que en este caso tiene como valor por defecto *Horizontal*.

5. Layout – DockPanel

Propiedad
adjunta

```
<DockPanel LastChildFill="True">
  <Button DockPanel.Dock="Top">Top Button</Button>
  <Button DockPanel.Dock="Bottom">Bottom Button</Button>
  <Button DockPanel.Dock="Left">Left Button</Button>
  <Button DockPanel.Dock="Right">Right Button</Button>
  <Button>Remaining Space</Button>
</DockPanel>
```

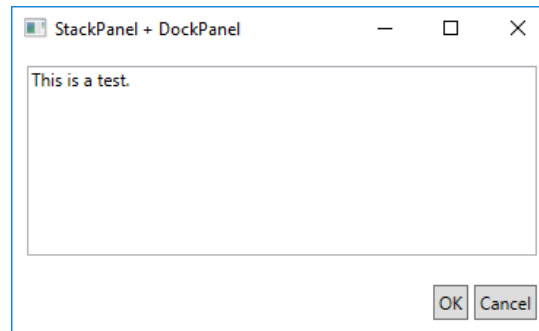


El *DockPanel* es un contenedor en el que los controles se pueden acoplar en alguna de las cuatro direcciones: arriba, abajo, derecha o izquierda.

Para indicar la dirección donde se debe acoplar un control se utiliza la propiedad *DockPanel.Dock*. Ésta es una propiedad adjunta, ya que se añade a un control por el hecho de estar contenido en un *DockPanel*.

Por defecto, el último control que incluyamos dentro del contenedor ocupará todo el espacio restante. Si queremos evitar este comportamiento, deberemos dar el valor *False* a la propiedad *LastChildFill* del *DockPanel*.

5. Layout – StackPanel + DockPanel



```
<DockPanel LastChildFill="True">  
  <StackPanel DockPanel.Dock="Bottom" HorizontalAlignment="Right" Orientation="Horizontal">  
    <Button Margin="10,10,2,10" Padding="3">OK</Button>  
    <Button Margin="2,10,10,10" Padding="3">Cancel</Button>  
  </StackPanel>  
  <TextBox Margin="10">This is a test.</TextBox>  
</DockPanel>
```

En esta diapositiva se muestra como se pueden combinar distintos tipos de paneles para conseguir el diseño deseado.

En este caso se está utilizando un *DockPanel* como contenedor principal, que tiene encajado en la zona inferior un *StackPanel* horizontal, para albergar a los botones. El cuadro de texto, al ser el último control del *DockPanel*, ocupará todo el espacio disponible al redimensionar la ventana.

5. Layout – Grid

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition></RowDefinition>
    <RowDefinition></RowDefinition>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
  </Grid.ColumnDefinitions>

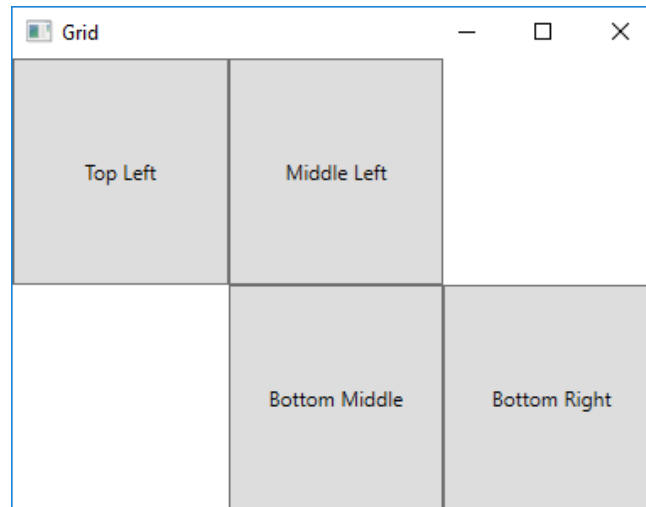
  <Button Grid.Row="0" Grid.Column="0">Top Left</Button>
  <Button Grid.Row="0" Grid.Column="1">Middle Left</Button>
  <Button Grid.Row="1" Grid.Column="2">Bottom Right</Button>
  <Button Grid.Row="1" Grid.Column="1">Bottom Middle</Button>
</Grid>
```

El *Grid* es uno de los contenedores más utilizados en WPF. Este panel nos permite estructurar el contenido en forma de tabla.

Como se puede apreciar en el ejemplo, lo primero que hacemos es definir cuantas filas y cuantas columnas tiene el *Grid*, mediante las propiedades *RowDefinitions* y *ColumnDefinitions*.

Para asignar un control en una de las celdas de la tabla, se usan las propiedades adjuntas *Grid.Row* y *Grid.Column*. El valor por defecto de estas propiedades es 0 (es decir, primera fila o primera columna).

5. Layout – Grid



Aquí podemos ver el resultado del código anterior. Como se puede apreciar en la imagen, si no se especifica el tamaño del control éste ocupará todo el espacio disponible.

5. Layout – Grid

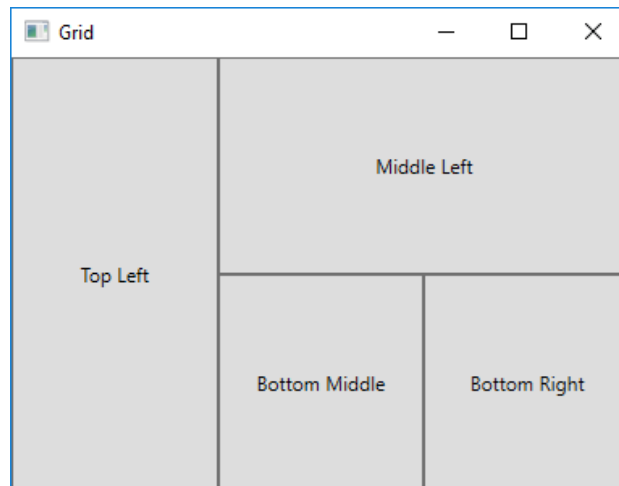
```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition></RowDefinition>
    <RowDefinition></RowDefinition>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
  </Grid.ColumnDefinitions>

  <Button Grid.Row="0" Grid.Column="0" Grid.RowSpan="2">Top Left</Button>
  <Button Grid.Row="0" Grid.Column="1" Grid.ColumnSpan="2">Middle Left</Button>
  <Button Grid.Row="1" Grid.Column="2">Bottom Right</Button>
  <Button Grid.Row="1" Grid.Column="1">Bottom Middle</Button>
</Grid>
```

Además de las propiedades para definir la fila y la columna en la que se situará el control, existen otras dos propiedades adjuntas que tendrán los controles situados en un *Grid*, *Grid.RowSpan* y *Grid.ColumnSpan*.

Estas dos propiedades permiten que un control se expanda en más de una fila o una columna.

5. Layout – Grid



Como se puede ver, el control situado en la celda 0,0 se está expandiendo dos filas, mientras que el situado en la celda 0,1 se está expandiendo dos columnas.

Este contenedor permite que varios controles ocupen la misma celda. En ese caso, los controles quedarán superpuestos, a no ser que desplazemos uno de ellos utilizando el margen.

5. Layout – Grid

- **Tamaño de filas y columnas:**

- Auto: tamaño necesario para alojar el contenido

```
<RowDefinition Height="Auto"></RowDefinition>
```

- Absoluto: tamaño fijo representado por un valor numérico

```
<RowDefinition Height="150"></RowDefinition>
```

- Proporcional: se reparte el espacio disponible de forma proporcional

```
<RowDefinition Height="*"></RowDefinition>
```

Es posible especificar el tamaño de las filas (alto) y las columnas (ancho) de un *Grid*. Para ello, se utilizan las propiedades *Height* y *Width* dentro de los elementos *RowDefinition* y *ColumnDefinition*, respectivamente.

A la hora de especificar el tamaño tenemos tres opciones:

- Auto: usando esta opción el tamaño de la fila o columna se ajustará al tamaño de su contenido
- Absoluto: podemos especificar un tamaño concreto por medio de un valor numérico
- Proporcional: utilizando la notación de estrella (*) se puede repartir el espacio de forma proporcional. Es el valor por defecto.

5. Layout – Grid

```
<Grid ShowGridLines="True" Width="400">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100"/>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="2*" />
  </Grid.ColumnDefinitions>

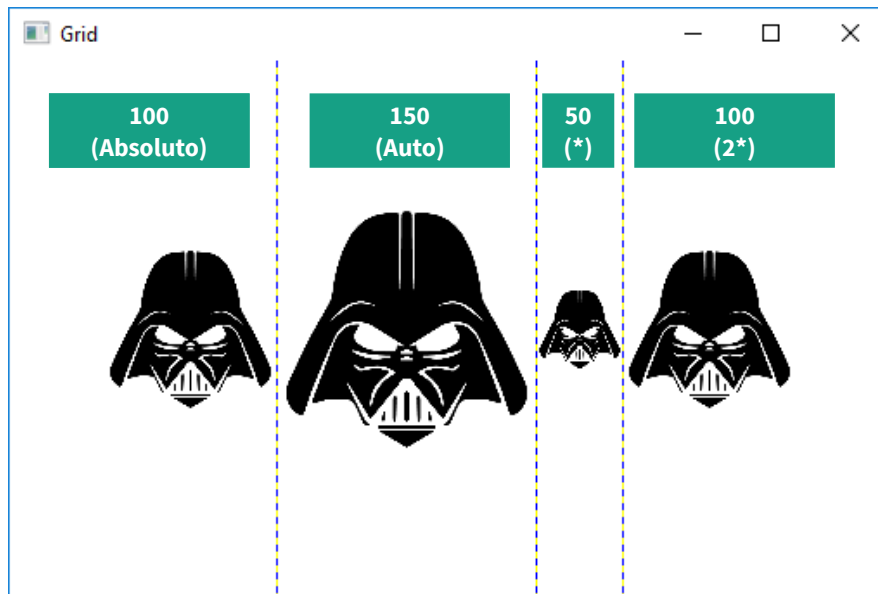
  <Image Height="150" Grid.Column="0" Source="vader.jpg"/>
  <Image Height="150" Grid.Column="1" Source="vader.jpg"/>
  <Image Height="150" Grid.Column="2" Source="vader.jpg"/>
  <Image Height="150" Grid.Column="3" Source="vader.jpg"/>
</Grid>
```

En este ejemplo de un *Grid* con una sola fila y cuatro columnas se ilustra el uso de las tres formas de especificar el tamaño.

La primera columna tiene un tamaño absoluto de 100. La segunda, sin embargo, se ha establecido en *Auto*, por lo que tendrá el ancho requerido por la imagen (en este caso 150). Las dos últimas columnas utilizan la notación proporcional, repartiendo el espacio restante en la proporción indicada (la cuarta columna tendrá el doble de ancho que la tercera).

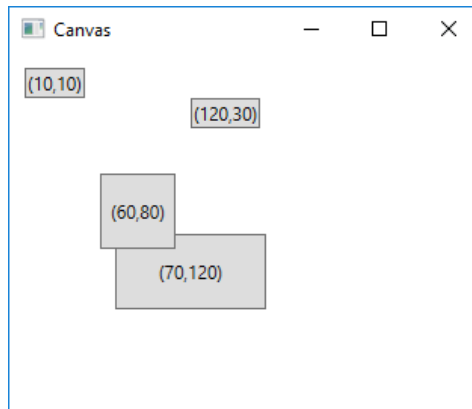
Destacar también que se está utilizando la propiedad *ShowGridLines* del *Grid* para mostrar líneas de división entre las celdas.

5. Layout – Grid



5. Layout – Canvas

```
<Canvas>
  <Button Canvas.Left="10" Canvas.Top="10">(10,10)</Button>
  <Button Canvas.Left="120" Canvas.Top="30">(120,30)</Button>
  <Button Canvas.Left="60" Canvas.Top="80" Width="50" Height="50" Canvas.ZIndex="1">(60,80)</Button>
  <Button Canvas.Left="70" Canvas.Top="120" Width="100" Height="50">(70,120)</Button>
</Canvas>
```

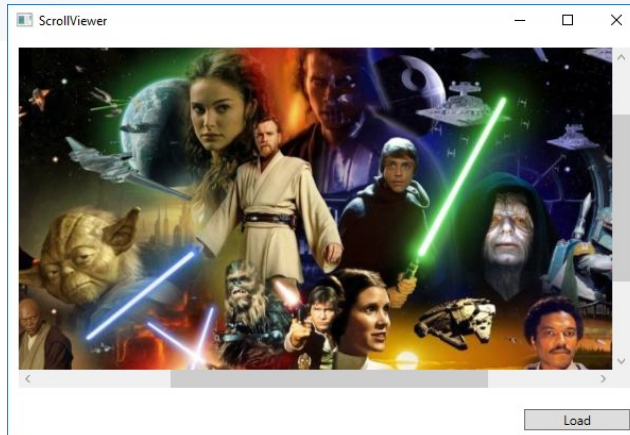


El último panel que vamos a ver es el *Canvas*. Este panel permite posicionar los controles de forma absoluta, indicando la distancia desde el extremo izquierdo y superior del contenedor al lado izquierdo y superior del control. Para ello utilizaremos las propiedades *Canvas.Left* y *Canvas.Top*.

Cuando varios controles están superpuestos, podemos controlar el orden mediante la propiedad adjunta *Canvas.Zindex*. Un valor más alto de esta propiedad indica que el control quedará por encima. El valor por defecto es 0.

5. Layout – ScrollViewer

```
<DockPanel LastChildFill="True" >  
  <StackPanel DockPanel.Dock="Bottom" HorizontalAlignment="Right" Orientation="Horizontal">  
    <Button Width="100" Margin="10">Load</Button>  
  </StackPanel>  
  <ScrollViewer VerticalScrollBarVisibility="Auto" HorizontalScrollBarVisibility="Auto" Margin="10">  
    <Image Source="starwars.jpg" Stretch="None" ></Image>  
  </ScrollViewer>  
</DockPanel>
```

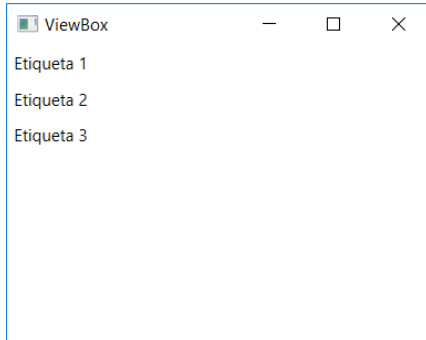


El ScrollViewer es un elemento que nos ayuda a disponer de barras de desplazamiento en cualquier control.

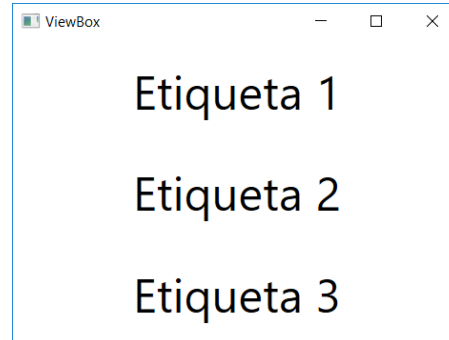
Con las propiedades *VerticalScrollBarVisibility* y *HorizontalScrollBarVisibility* podemos controlar el comportamiento de este control. El valor *Auto* (por defecto) hará que las barras de desplazamiento se muestren solo cuando sea necesario. También podemos configurar el control para que siempre muestre las barras, o para que no las muestre aunque sea necesario para acceder a todo el contenido.

5. Layout – ViewBox

```
<StackPanel>  
  <Label>Etiqueta 1</Label>  
  <Label>Etiqueta 2</Label>  
  <Label>Etiqueta 3</Label>  
</StackPanel>
```



```
<Viewbox>  
  <StackPanel>  
    <Label>Etiqueta 1</Label>  
    <Label>Etiqueta 2</Label>  
    <Label>Etiqueta 3</Label>  
  </StackPanel>  
</Viewbox>
```



Este es el último control que veremos en el apartado de *layout*. El *ViewBox* es un control de tipo decorador (como el *Border*) que permite escalar visualmente su contenido.

En el ejemplo se muestra un *StackPanel* con tres etiquetas, comparando el resultado de usar o no un *ViewBox*. Como se puede apreciar en el ejemplo de la derecha, al utilizar el *ViewBox* las tres etiquetas se escalan para ocupar todo el espacio disponible en el *ViewBox*.

6. Creación dinámica de controles

```
public MainWindow()
{
    InitializeComponent();

    // Creamos y configuramos dos etiquetas
    Label etiqueta1 = new Label();
    etiqueta1.Content = "Etiqueta 1";
    Label etiqueta2 = new Label();
    etiqueta2.Content = "Etiqueta 2";

    // Asociamos a las etiquetas un manejador de eventos
    etiqueta1.MouseEnter += new MouseEventHandler(manejador);
    etiqueta2.MouseEnter += new MouseEventHandler(manejador);

    // Creamos un StackPanel y le añadimos las etiquetas
    StackPanel contenedor = new StackPanel();
    contenedor.Children.Add(etiqueta1);
    contenedor.Children.Add(etiqueta2);

    // Establecemos el StackPanel como contenido de la ventana
    this.Content = contenedor;
}
```

Aunque hasta este momento hemos utilizado el lenguaje XAML para definir la estructura de la interfaz de usuario de la aplicación, también podemos crear y configurar los controles desde el código trasero. Esta funcionalidad nos permitirá modificar la interfaz de forma dinámica durante la ejecución de la aplicación.

En el ejemplo se están creando los controles en el constructor de la ventana. Inicialmente se crean dos etiquetas y se les asignan propiedades y manejadores de eventos. A continuación se crea un *StackPanel* y añadimos en su interior las etiquetas.

Por último, se establece el *StackPanel* como contenido de la ventana.

7. Estilos

```
<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfApplication1"
        mc:Ignorable="d"
        Title="EStilos" Height="327.39" Width="433.657">

    <Window.Resources>

        <!-- Estilo para todos los botones -->
        <Style TargetType="{x:Type Button}">
            <Setter Property="Width" Value="100"/>
            <Setter Property="Height" Value="30"/>
            <Setter Property="FontFamily" Value="Tahoma"/>
            <Setter Property="FontSize" Value="18"/>
        </Style>

        <!-- Estilo para todas las etiquetas -->
        <Style TargetType="{x:Type Label}">
            <Setter Property="FontFamily" Value="Tahoma"/>
            <Setter Property="FontSize" Value="20"/>
        </Style>

    </Window.Resources>

    <Grid>
```

De forma análoga a las hojas de estilo CSS en el desarrollo web, los estilos de WPF nos permiten agrupar la configuración de ciertas propiedades para poder reutilizarlas fácilmente.

Aunque veremos que existen otras posibilidades, en principio definiremos los estilos dentro de la propiedad *Resources* del elemento *Window*. Por cada uno de los estilos que queramos definir tendremos un elemento *Style*. Por medio de su propiedad *TargetType* indicaremos el tipo de control al que va destinado.

Dentro del elemento *Style* utilizaremos elementos *Setter* para definir la propiedad y el valor que se está asignando en el estilo.

7. Estilos

```
<Window.Resources>

<!-- Estilo para todos los botones -->
<Style TargetType="{x:Type Button}">
    <Setter Property="Width" Value="100"/>
    <Setter Property="Height" Value="30"/>
    <Setter Property="FontFamily" Value="Tahoma"/>
    <Setter Property="FontSize" Value="18"/>
</Style>

<!-- Estilo para todas las etiquetas -->
<Style TargetType="{x:Type Label}">
    <Setter Property="FontFamily" Value="Tahoma"/>
    <Setter Property="FontSize" Value="20"/>
</Style>

</Window.Resources>
```

Extensión
de marcado
{...}

En esta diapositiva se muestra con más detalle la parte del código destinada a definir los estilos.

En el ejemplo se está definiendo un estilo para los botones (que afectará a todos los botones contenidos en esta ventana) y otro para las etiquetas (que de la misma forma, afectará a todas las etiquetas).

Si el valor de alguna de las propiedades que queremos asignar en el estilo no se puede establecer como atributo XML (con una cadena de texto) se debería usar la sintaxis de subelemento XML que vimos al inicio del tema.

7. Estilos - Explícitos

```
<Window.Resources>

    <!-- Estilo para todas las etiquetas -->
    <Style TargetType="{x:Type Label}">
        <Setter Property="FontFamily" Value="Comic Sans MS"/>
        <Setter Property="FontSize" Value="20"/>
    </Style>

    <!-- Estilo para todas las etiquetas de tipo Titulo-->
    <Style TargetType="{x:Type Label}" x:Key="Titulo">
        <Setter Property="FontSize" Value="40"/>
        <Setter Property="FontWeight" Value="Bold"/>
    </Style>

</Window.Resources>

<StackPanel>
    <Label Style="{StaticResource Titulo}">Titulo</Label>
    <Label>Texto normal</Label>
</StackPanel>
```

También tenemos la posibilidad de definir un estilo que no se aplique a todos los controles de un tipo, si no solamente a los que nosotros indiquemos. Para ello, utilizaremos la propiedad *x:Key* del elemento *Style* para asociarle un nombre.

En el ejemplo se ha modificado el estilo asociado a las etiquetas para que no aplique a todas las etiquetas por defecto. Destacar que de cualquier modo tenemos que seguir indicando el tipo de control al que va destinado el estilo.

Para asignar el estilo a un control concreto, utilizaremos la propiedad *Style* del control. Y para hacer referencia al estilo la extensión de marcado *StaticResource*.

7. Estilos – Herencia de estilos

```
<Window.Resources>

    <!-- Estilo para todas las etiquetas -->
    <Style TargetType="{x:Type Label}">
        <Setter Property="FontFamily" Value="Comic Sans MS"/>
        <Setter Property="FontSize" Value="20"/>
    </Style>

    <!-- Estilo para todas las etiquetas de tipo Titulo-->
    <Style TargetType="{x:Type Label}" x:Key="Titulo" BasedOn="{StaticResource {x:Type Label}}">
        <Setter Property="FontSize" Value="40"/>
        <Setter Property="FontWeight" Value="Bold"/>
    </Style>

</Window.Resources>

<StackPanel>
    <Label Style="{StaticResource Titulo}">Titulo</Label>
    <Label>Texto normal</Label>
</StackPanel>
```

Otra de las posibilidades que nos ofrece WPF es la definir un estilo a partir de otro ya creado. Es lo que se conoce como herencia de estilos.

Para ello, el estilo que hereda deberá indicarlo en la etiqueta *Style* con la propiedad *BasedOn*.

En el ejemplo se muestra un estilo genérico para etiquetas y uno explícito que heredará de él. Si heredáramos de otro estilo explícito se utilizaría directamente su nombre junto a *StaticResource*, sin necesitar la extensión de marcado anidada.

7. Estilos

```
<!-- Estilo con propiedad compleja -->
<Style TargetType="{x:Type Label}">
    <Setter Property="FontFamily" Value="Comic Sans MS"/>
    <Setter Property="FontSize" Value="20"/>
    <Setter Property="Effect">
        <Setter.Value>
            <DropShadowEffect/>
        </Setter.Value>
    </Setter>
</Style>

<!-- Estilo con manejador de eventos -->
<Style TargetType="{x:Type Button}">
    <Setter Property="Width" Value="100"/>
    <Setter Property="Height" Value="30"/>
    <EventSetter Event="Click" Handler="MetodoManejador" />
</Style>

// Asignar dinámicamente un estilo desde el código
etiqueta1.Style = (Style) Resources["estilo"];
```

En esta diapositiva vemos algunas cuestiones extra relacionadas con los estilos.

En el primer ejemplo, vemos como asignar un valor complejo a una propiedad en un estilo, usando la sintaxis de subelemento XML.

En el segundo, vemos como es posible asociar en un estilo el manejador de un evento. Para ello, se utiliza el elemento *EventSetter*.

Por último, vemos como podríamos asociar por código un estilo a un control.

7. Estilos – Ámbito

```
<TextBlock Text="Style test">
  <TextBlock.Style>
    <Style>
      <Setter Property="TextBlock.FontSize" Value="36" />
    </Style>
  </TextBlock.Style>
</TextBlock>
```

Control

```
<StackPanel Margin="10">
  <StackPanel.Resources>
    <Style TargetType="TextBlock">
      <Setter Property="Foreground" Value="Gray" />
      <Setter Property="FontSize" Value="24" />
    </Style>
  </StackPanel.Resources>
  <TextBlock>Header 1</TextBlock>
  <TextBlock>Header 2</TextBlock>
  <TextBlock Foreground="Blue">Header 3</TextBlock>
</StackPanel>
```

Panel

Para finalizar con el apartado de estilos, vamos a ver que, además de en la sección *Resources* de la ventana, es posible definir estilos en otras partes de nuestro código XAML.

En el primer ejemplo vemos como podemos definir un estilo únicamente para un control, utilizando la sintaxis de subelemento para la propiedad *Style*.

En el segundo caso vemos como podemos definir los estilos para todos los controles contenidos en un panel, utilizando la propiedad *Resources* (de forma análoga a como lo hemos hecho hasta ahora para la ventana).

7. Estilos – Ámbito

```
<Window.Resources>
  <Style TargetType="TextBlock">
    <Setter Property="Foreground" Value="Gray" />
    <Setter Property="FontSize" Value="24" />
  </Style>
</Window.Resources>
```

Ventana

```
<Application.Resources>
  <Style TargetType="TextBlock">
    <Setter Property="Foreground" Value="Gray" />
    <Setter Property="FontSize" Value="24" />
  </Style>
</Application.Resources>
```

Aplicación
(App.xaml)

La primera opción que vemos en esta diapositiva es la que hemos utilizado en todos los ejemplos anteriores, es decir, definir los estilos a nivel de ventana y que, por lo tanto, apliquen a todos los controles de la misma.

Por último, sería posible definir los estilos a nivel de aplicación, lo que resultaría de utilidad en el caso de aplicaciones con múltiples ventanas. Para ello, incluiremos los estilos en la propiedad *Resources* del elemento *Application*, que podemos encontrar en el fichero *App.xaml* de nuestro proyecto.

7. Estilos – Ámbito

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                    xmlns:local="clr-namespace:WpfApp1">
    <Style TargetType="TextBox">
        <Setter Property="Foreground" Value="Gray" />
        <Setter Property="FontSize" Value="24" />
    </Style>
</ResourceDictionary>
```

**Diccionario
de recursos
(fichero XAML
independiente)**

```
<Window.Resources>
    <ResourceDictionary Source="Dictionary1.xaml" />
</Window.Resources>
```

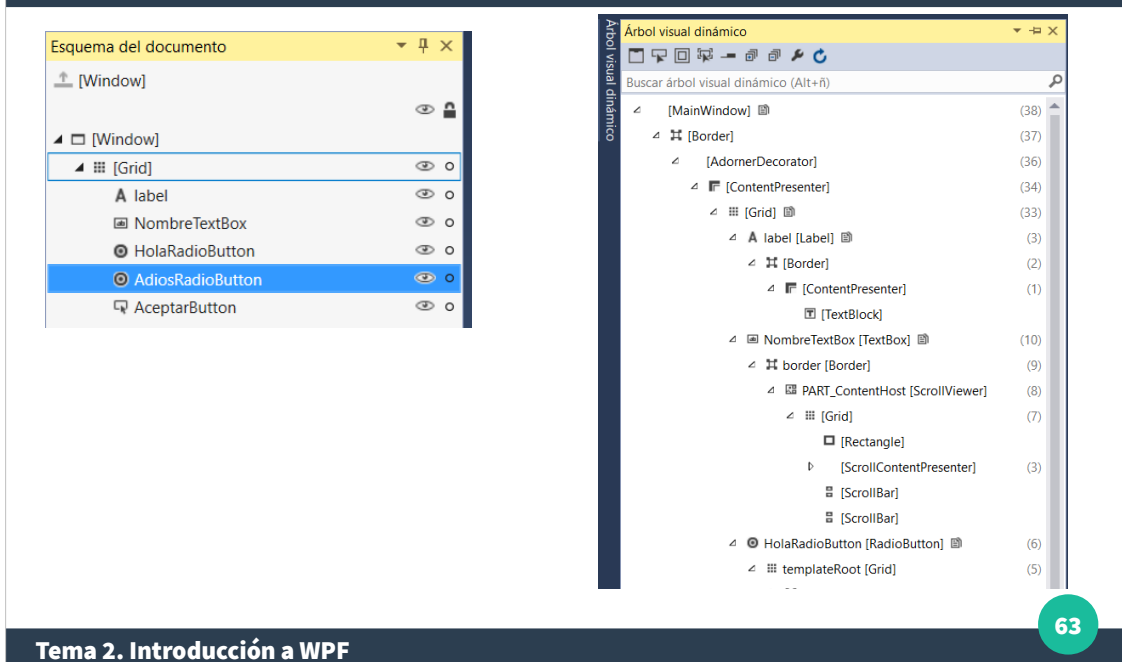
Uso del diccionario

Otra opción interesante, sobre todo en proyectos grandes es la de definir los estilos en un fichero independiente, llamado Diccionario de Recursos. Esta opción facilitaría la reutilización de los estilos de unas aplicaciones a otras.

Visual Studio permite crear diccionarios de recursos, y dentro de ellos se pueden situar directamente los estilos.

Para utilizarlos, se utiliza el elemento *ResourceDictionary*, indicando en su propiedad *Source* el fichero que contiene el diccionario,

8. Árbol lógico y árbol visual



Para acabar con este tema introductorio, vamos a comentar el concepto de árboles que existe en WPF.

Por un lado, tenemos un árbol lógico, que es el que queda definido por la estructura de elementos XML definida en el fichero XAML. Este árbol puede consultarse mediante la ventana “Esquema del documento” de Visual Studio.

Por otro lado, en WPF se habla de un árbol dinámico, que puede consultarse cuando se ejecuta la aplicación en la ventana “Árbol visual dinámico”. En este árbol se detallan los controles que componen internamente otros controles. Veremos este tema en más profundidad en el tema dedicado a las plantillas.

8. Árbol lógico y árbol visual

- Propiedades
 - Children / Child / Content / Items
 - Parent
 - Método FindName (recursivo)
 - Clases LogicalTreeHelper y VisualTreeHelper
- Árbol lógico

WPF nos ofrece algunas propiedades y métodos en los controles para movernos por el árbol lógico. Por ejemplo, tendremos una propiedad para acceder al hijo o hijos del control (que en función del tipo de control podrá ser *Children*, *Child*, *Content* o *Items*). También disponemos de la propiedad *Parent* para acceder al control padre en el árbol lógico.

El método *FindName* permite buscar un control por su nombre en los descendientes de un determinado control.

Por último, destacar que WPF ofrece dos clases estáticas con métodos para trabajar en ámbos árboles: *LogicalTreeHelper* y *VisualTreeHelper*.